Efficiency of Projectional Editing: A Controlled Experiment

Thorsten Berger
Chalmers | University of Gothenburg, Sweden

Markus Völter
independent / itemis Stuttgart, Germany

Hans Peter Jensen, Taweesap Dangprasert
IT University of Copenhagen, Denmark

Janet Siegmund
University of Passau, Germany

ABSTRACT

Projectional editors are editors where a user's editing actions directly change the abstract syntax tree without using a parser. They promise essentially unrestricted language composition as well as flexible notations, which supports aligning languages with their respective domain and constitutes an essential ingredient of model-driven development. Such editors have existed since the 1980s and gained widespread attention with the Intentional Programming paradigm, which used projectional editing at its core. However, despite the benefits, programming still mainly relies on editing textual code, where projectional editors imply a very different—typically perceived as worse—editing experience, often seen as the main challenge prohibiting their widespread adoption. We present an experiment of code-editing activities in a projectional editor, conducted with 19 graduate computer-science students and industrial developers. We investigate the effects of projectional editing on editing efficiency, editing strategies, and error rates—each of which we also compare to conventional, parser-based editing. We observe that editing is efficient for basic-editing tasks, but that editing strategies and typical errors differ. More complex tasks require substantial experience and a better understanding of the abstract-syntax-tree structure—then, projectional editing is also efficient. We also witness a tradeoff between fewer typing mistakes and an increased complexity of code editing.

CCS Concepts

•Software and its engineering → Integrated and visual development environments;

Keywords

projectional editing, language workbench, experiment

1. INTRODUCTION

Projectional editor describes a type of editor where users work on a projection of a program's abstract syntax tree (AST) and directly change the AST with their editing gestures. This concept is different from parser-based editing, where users change the concrete syntax (characters in a text buffer), and a parser then matches the syntax against a grammar definition to construct the AST. Projectional editing, also known as structured editing or syntax-directed editing, is not a new idea; early references go back to the 1980s and include the Incremental Programming Environment [32], GANDALF [35], and the Synthesizer Generator [39]. Work on projectional editors continues today: Intentional Programming [44, 18, 45, 14] is its most well-known incarnation. Other contemporary tools [20] are the Whole Platform [9], Más [3], Onion, and MPS [4]. The latter is the instrument of this work. Most of these projectional editors are used in language workbenches—tools for developing and composing languages [20, 21].

Projectional editors have two main advantages, both resulting from the absence of parsing. First, they support notations that cannot easily be parsed, such as tables, diagrams or mathematical formulas—each of which can be mixed with the others and with textual notations [45, 51]. Second, they support various ways of language composition [19], typically including modular language extension as well as embedding of unrelated languages into a host language [44, 50]. Projectional editors can deal with mixed-language code while retaining awareness of the code structure (syntactic ambiguities are avoided), which is much harder to achieve with parser-based tools. This is crucial for supporting analysis, transformation, and meaningful IDE support in the workbenches.

These two advantages are the essential ingredients for realizing domain-specific languages or to add domain-specific abstractions to existing, general-purpose languages. Domain specificity, in turn, is a major contributor to model-driven development as well as productivity in software engineering in general, as reported for domains as diverse as language and compiler implementation [22, 25], embedded software [13, 27, 29, 30, 56], and web applications [48]. In fact, developers commonly appreciate the ability for language composition and notational flexibility [24, 41, 55].

These benefits come at a cost. Even though projectional editors support a wide range of non-textual notations, a significant share of any program, such as expressions and statements, will be expressed textually. Thus, it is crucial that projectional editors do not negatively impact editing efficiency for textual notations. This is their weak spot: for textual notations, projectional editors behave differently from what developers know from traditional text editors in terms of the granularity and restrictions of code edits and code selections. These differences are perceived as a problem [37].

FSE'16, November 13–18, 2016, Seattle, WA, USA
ACM. 978-1-4503-4218-6/16/11...$15.00
http://dx.doi.org/10.1145/2950290.2950315

Early projectional editors from the 1980s did very little to address this issue, ultimately limiting their adoption. Contemporary tools, such as MPS, have significantly improved usability, but inherited the bad reputation. However, no empirical data on their use is available.

In this paper, we address this gap by studying how developers perform common code-editing activities in a projectional editor. We analyze the effects of projectional editing on editing efficiency, strategies, and types and frequencies of errors made. We also determine how the use of projectional editing differs from conventional, parser-based program editing, whether experience helps, and whether projectional editing successfully abstracts from its underlying technicalities (e.g., AST structure) or whether a deeper understanding is indispensable. By identifying hotspots that are problematic for adoption and efficient editing, we support tool vendors and researchers to provide better projectional editors.

Our study comprises an experiment with 19 participants, designed based on the qualitative results of a survey we conducted before with professional developers who are familiar with projectional editing [55]. Our experiment comprised two phases: a controlled experiment with 14 computer-science (CS) students and a quasi experiment with five industrial developers experienced with projectional editing. All performed the same set of code-editing activities. Our instrument was JetBrains MPS, since (i) it is the most widely used projectional editor today, (ii) it improved significantly over the tools from the 1980s, warranting a new look at editing efficiency, and (iii) it is open-source software, which fosters replicability of our results.

In summary, we contribute: (i) qualitative and quantitative results about the use of projectional editing from an experiment conducted with student and industrial participants, and (ii) a replication package in an **online appendix** [8].

2. BACKGROUND

Projectional Editing and MPS. Projectional editors avoid parsing the concrete syntax to build a program's AST. Instead, editing activities by a user *directly* change the AST. The user sees and interacts with a representation of the program rendered by projection rules that reflect the AST as it changes. This approach is fundamentally similar to graphical editors (e.g., UML tools), but projectional editors generalize the approach to arbitrary notations, including textual ones.

Code completion plays a critical role: As the user picks something from the code-completion menu, the selected language concept is instantiated and added to the AST. Text strings called *aliases* are used to pick language constructs from the menu, and the menu contents are driven by the language definition. Parsing is avoided, because *every single next string is recognized as it is entered*, no token structure has to be recognized in a token stream. Disambiguation is performed by the user at the time of picking a concept from the code-completion menu and not by a parser (based on a potentially complex structure). After a user has picked a language construct and it has been instantiated in the AST, a program is *never* ambiguous: every node points to its defining concept. This is important, because independently developed languages can be composed in a single program and never lead to structural or syntactic problems, *irrespective of the concrete syntax* of the participating languages. For instance, two languages could have overlapping keywords, which parser-based tools could not easily disambiguate (without the need for writing dedicated disambiguation code for the combination of the two languages).

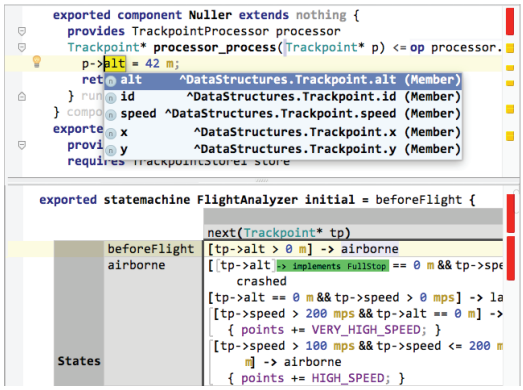


Figure 1: MPS IDE with mbeddr languages. Top: textual notation for software components. Bottom: tabular notation for state machines (a graphical one also exists [6]) mixed with textual code and with requirements trace labels.

Every AST node has a unique ID. References between nodes (e.g., variable use and variable definition) are references between node IDs—created by selecting the reference target from the code-completion menu. This is in contrast to parser-based editors, where references are strings that capture the (qualified) name of the target node, and a subsequent phase performs name matching and reference resolution. Programs are typically persisted by serializing the AST (e.g., using XML) to avoid re-parsing when a program is loaded again.

The JetBrains Meta Programming System (MPS) [4] is an open-source language workbench [20], that is, a system for defining, composing, and using languages and their IDEs. It relies on a projectional editor and supports concrete and abstract syntax, type systems, and transformations, as well as IDE aspects, such as syntax highlighting, code completion, find-usages, diff/merge, refactoring, and debugging. MPS' projectional editor enables the aforementioned flexible notations and language composition.

Adoption in Industry and mbeddr. Although being researched since the 1980s, projectional editors have not been widely adopted in industry, including those we discuss in Sec. 7. They are niche tools, mainly due to their usability problems. However, given their potential and their powerful facilities, the motivation for our work is to leverage existing research results by studying and eventually improving their code-editing efficiency. Despite the low adoption in general, the use of MPS in particular has been growing significantly over the last years; it is used in production for various applications, such as JetBrains YouTrack, computational biology [43], web applications (Code Orchestra IDE [2]), requirements engineering [54], insurance DSLs, security assessments, satellite control software, cloud-based business applications as well as health and medicine applications. A migration of an existing legacy language infrastructure for the banking domain into MPS is also reported [31].

The biggest application of MPS is mbeddr, a set of integrated languages for embedded software engineering [5, 52], developed with MPS. Its core is an extensible version of the C programming language (C99) plus extensions for interfaces and components, state machines, and physical units, among others. mbeddr provides multi-paradigm programming for C [17], in which different abstractions and notations can be used in the same program, as shown in Fig. 1, which illus-

trates the facilities for language composition and flexible notations. The language extensions are not mere libraries or frameworks; they are first-class language constructs with full support from the compiler, type system, and IDE. mbeddr also supports languages for cross-cutting concerns, such as documentation, requirements management, and traceability, as well as product-line engineering. Several formal verification techniques are also directly integrated with the languages [34, 38]. mbeddr comprises 70 languages, 35 of them are C extensions: it is one of the largest systems built on top of a language workbench. mbeddr is open-source software and used in several commercial embedded-software projects (e.g., the smart meter [57]).

Preparatory Survey. Previously, we surveyed 21 industrial developers using projectional editing mainly in the embedded and automotive domain. We presented the survey before [55], with a focus on a quantitative analysis of the questionnaire's closed, Likert-scale questions, which elicited the participants' perceptions of projectional editing. To prepare our experiment, we additionally analyzed the questionnaire's remaining open questions, which elicited qualitative experiences from the participants. Using open coding [46], we identified editing-related aspects in four categories: *basic editing*, *errors*, *AST conformance*, and *refactoring*. Our results show that basic editing was not considered as a problem, except for some over-deletions and issues with editing structures that crosscut the AST (e.g., parentheses). Still, participants indicated that some common editing behaviors they are used to were not applicable. We also observe a tendency towards fewer errors; for instance, some respondents conjectured less syntactic errors due to the permanently enforced AST conformance. Yet, AST conformance in general was perceived negatively—developers wanted more freedom. They were especially negative about editing expressions (e.g., `a+b*(c+f)`) and the fact that references can only be established once the reference target is available. Finally, participants expressed that refactorings work like in regular IDEs, except the rename refactoring, which works automatically and everywhere, because of a projectional editor's reliance on unique IDs (and not names). These results guided our experiment design.

3. EXPERIMENT DESIGN

We conducted the experiment in two phases, one with graduate CS students (controlled experiment) and one with industrial participants experienced with MPS (quasi experiment). We now discuss our research questions, the detailed experiment objective, the tasks, and our materials. We also describe participants, experiment execution, and the analysis method. We discuss confounding parameters when relevant.

3.1 Research Questions

Our first research question aimed at quantitatively comparing editing efficiencies of projectional and parser-based editors. It investigated whether one is faster, for what editing tasks, and what the effects of using projectional editing are on beginners.

RQ1 **How does editing efficiency differ between projectional and parser-based editors?** We investigated this question in the first phase with a controlled experiment and students who were inexperienced with projectional editing. We randomly grouped them into a projectional-editor and a parser group, and then analyzed their efficiencies.

To enhance the external validity of our results, and to obtain insights into whether substantial experience has an effect on the use of projectional editing, we conducted the experiment again three weeks later with industrial participants experienced with MPS, who were not available for the first phase. We quantitatively investigated:

Table 1: Experiment design. Top: first phase with students. Bottom: second phase with experienced MPS users.

editor	experience with MPS	group	# of part.
projectional	beginner	**Proj**	8
parser	–	**Par**	6
projectional	expert	**ProjE**	5

RQ2 **What is the editing efficiency of experienced projectional-editing developers?** This second phase also allowed us investigating whether and how projectional editing can outperform parser-based editing. Since these participants were not randomly split into a projectional and a parser group, this second phase is a quasi experiment [42]. Although we could assume that our CS-student and industrial participants are equally proficient with textual editing (i.e., using the keyboard), we separate RQ2, which aimed at cross-phase quantitative results, from the discussion of RQ1 and RQ3.

Our third research question was more exploratory. After the two phases, we wanted to understand the detailed usage of how editing differs among all the groups in more detail:

RQ3 **What are commonalities and differences in the use of projectional and parser-based editors?** We investigated this research question by conducting a fine-grained analysis of the code-editing tasks across both phases. We compared all groups by determining editing operations and effects (e.g., errors), identifying and comparing editing strategies, and eliciting our participants' perceptions and experiences.

Note the different validities of quantitatively investigating the editing efficiency in the controlled experiment (RQ1) and in the quasi experiment (RQ2). The validity of the latter is slightly lower, since we could not control for any differences in textual-editing proficiency of the industrial developers, as opposed to the students. Thus, we added RQ3 to complement the quantitative analysis (hypothesis testing) and provide important empirical data on the detailed use of projectional editing by the different groups of participants.

3.2 Objective: Variables and Hypotheses

We defined the following variables of interest and formulated the following hypotheses about their relation.

3.2.1 Variables

For the first phase we have one independent variable, `editor`, which has two levels, `projectional` and `parser-based`. The second phase introduces another, quasi-independent variable `experience with projectional editing` with the two levels `expert` (professional experience with projectional editing) and `beginner` (no prior use of projectional editing). Note that *programming* experience is neither an independent variable nor a confounding factor, since the tasks did only require basic programming skills (discussed in Sec. 3.3 and Sec. 6).

The two student groups we formed each correspond to one of the two levels of the independent variable `editor`. With respect to the second independent variable `experience with projectional editing` they are `beginner`. In the second phase, the industrial participants form one group, which corresponds to the level `projectional` of the independent variable `editor`, and the level `expert` of the independent

variable `experience with projectional editing`.

Table 1 shows the resulting experiment design with the three groups. In the remainder, for brevity, we will use the abbreviations Proj (for `projectional-beginner`), Par (for `parser`), and ProjE (for `projectional-expert`). We used a fractional experiment design, because we could assume that all participants have comparable skills in text editing and it was difficult to find more experts to create an additional control group. A few students also did not show up. We compensated the slightly different group sizes by using robust statistical significance tests in our quantitative analysis.

The dependent variable is `efficiency`, measured quantitatively using the metric `completion time` of the tasks (defined in Sec. 3.5). The other dependent variables are `use of operations`, `editing errors`, and `editing strategies`, analyzed mostly qualitatively from the experiment recordings. A post-experiment questionnaire and debriefing interviews complemented all these data sources.

3.2.2 Hypotheses

We defined hypotheses using the results of our preparatory survey, our experiences, and common claims about projectional editors in the literature. Specifically, the survey allowed us to state one-tailed hypotheses to increase the statistical power, which is important for our small sample size.

RQ1. For the first research question, we ran the controlled experiment with the student participants, hypothesizing:

H1. Par *is more efficient than* Proj.

Our suspicion was that the different editing experience slows down beginners, even though we provide a 45-minute MPS tutorial (explained shortly in Sec. 3.4) before the experiment.

RQ2. For this question, we ran the quasi experiment and analyzed the editing performance across all the groups using the following two comparisons and hypotheses. We first compared ProjE with Proj, which would show us whether experience can affect projectional editing. We hypothesized:

H2. ProjE *is more efficient than* Proj.

Thereafter, we compared ProjE with Par, hypothesizing that:

H3. ProjE *is more efficient than* Par.

This hypothesis conjectured that developers can become more efficient with a projectional editor than those with a parser-based editor. It would mean that the benefits provided by projectional editors are not hindered by reduced editing efficiency after training is provided.

RQ3. This question is more exploratory, comparing projectional and parser-based editing in detail. So we did not formulate a hypothesis, but conjectured differences how the editing tasks are solved and what kinds of editing patterns and effects (e.g., mistakes) arise. Recall the results of our preceding survey, that many behaviors known from parser-based editing are not applicable in projectional editing, which we conjectured leads to different editing strategies.

3.3 Material and Tasks

Tools. For group Par, we used Eclipse CDT as a common parser-based editor. For Proj and ProjE, we used a standard installation of MPS and provided tasks in the mbeddr language. For screen recordings, we used CamStudio [1]. Every participant received a startup package (fully configured project with imported models/code, build infrastructure, and test cases) with code templates for the required tasks. We also provided a cheat sheet with common keyboard shortcuts.

Questionnaire and Debriefing Interviews. After each phase, we used a questionnaire to elicit the participants' perceptions. Closed questions had a five-point Likert scale (e.g., "I would solve this problem differently in my favorite 'conventional' editor.") with options ranging from strongly disagree to strongly agree. Open questions asked about qualitative experiences (e.g., "Was there anything you found particularly difficult with refactorings?") or were inspired by our survey (e.g., "When working with the editor, I often deleted more than I expected."). We also designed debriefing interviews with three main questions: "What were the biggest problems?", "How did you solve them?", and "What do you think in general about MPS and the editing approach?"

Tasks. The tasks reflected common code-editing activities and covered aspects identified in the survey. All tasks were based on C. Since mbeddr implements C99 with very few changes [53], the code was fully identical for all groups. We designed the tasks to cover both basic (e.g., insertion, deletion) and advanced (e.g., move, refactoring) editing facilities, structured according to the four categories from the survey (*basic editing*, *errors*, *AST conformance*, and *refactoring*):

Task 1) Edit **expressions** that represent logical laws (double negation law, commutative law, associative law, De Morgan's law, absorption law): It investigated fine-grained editing and the ability to perform cross-tree changes (adding nodes consistently at different tree locations; e.g., inserting parentheses).

Task 2) Implement a **bubble-sort algorithm**: This task analyzed the use and composition of common program elements, such as `if` and `for` statements, as well as function and variable declarations in a typical programming activity.

Task 3) Implement **function signatures** (headers): This task investigated how users deal with the always enforced correct AST—more precisely, how they handle cross references where the target is not yet defined. Recall that projectional editors establish reference targets at editing time.

Task 4) Perform **refactorings**: This task investigated strategies used to modify and refactor code, and to what extent IDE-supported refactoring methods (e.g., rename, extract method, inline variable, extract constant) are used in addition to basic-editing operations (e.g., *CopyPaste*).

To avoid bias originating from varying programming abilities, we gave the solution to the participants. This is valid since we evaluated editing efficiency, not programming.

For the first task, we provided the laws in the editor, but introduced an error. Participants were asked to enter the correct solution. For instance, negations in the De Morgan's law were missing, and the correct solution was to insert them:

```
boolean deMorgansLaw = p && q == p || q;         /* Incorrect */
boolean deMorgansLaw = !(p && q) == !p || !q;    /* Correct */
```

This case required cross-tree editing operations not supported in the used MPS version. For the second task, we provided a solution and instructed participants to enter it in the editor. In the third task, five method signatures representing the API of a book library (including functions, such as `searchBook()` and `reserveBook()`) had to be created. Participants also had to create the respective parameter and return types, where the task description explained that empty `structs` can be used, so writing `struct NAME {}` sufficed, which is also proposed by MPS' code-completion when typing `struct`. In the fourth task, participants had to solve three complex code modification and refactoring tasks. In each, we provided the implementation of a function that handles the order process of an online shop. Participants had to conceive strategies

and apply one or more simple refactoring operations (either IDE-provided ones or *CopyPaste*, *CutPaste*, and so on) to achieve a complex refactoring, comprising method extraction, flattening of deeply nested `if` statements, or inlining of variables. The subtasks were in fact very complex (cf. the task descriptions [8]) and none could be solved by only applying a single IDE-provided refactoring.

3.4 Participants and Experiment Execution

For the first phase, we recruited 14 graduate CS students through flyers and mailing lists and randomly assigned them to the Proj and Par group. For the second phase, we recruited five developers from an industrial partner, where they regularly used projectional editing. Although not directly relevant for the experiment (only basic programming skills required), the students had been programming for 1 to 2 years, the professionals for 5 to 10 years. Most of the latter had used projectional editors for at least one year (one even more than three years), two had been using it for less than a year. For compensation, the students could enter a raffle for vouchers (for an electronics store, cinema, or coffee shop).

We conducted the first phase on two appointments for Proj and Par. In each, we introduced our study and explained projectional editing. For Proj, we also provided a 45-minutes hands-on tutorial on MPS/mbeddr. For Par, we recapitulated Eclipse CDT. We showed them how to run the test cases (provided for every task)—once these passed, participants should proceed to the next task. Thereafter, participants solved the tasks with instructions provided on paper. Finally, they completed the questionnaire and volunteers participated in the debriefing interviews. The second phase was conducted with ProjE three weeks later and followed exactly the same procedure, except that we could skip the MPS tutorial.

3.5 Analysis

We measured the completion time by identifying the first and last edit a user performed on a task from the screen recordings. So completion time excludes reading the task description, testing, and building the project. For all participants and tasks, we verified that the task was completed, making a subjective assessment whether the solution was structurally correct with only few mistakes, which we analyzed separately. The metric allowed us to test our hypotheses using the statistical tests ANOVA and Kruskal-Wallis [10].

We then analyzed the screen recordings in-depth. For the basic-editing tasks, we identified editing operations and editing effects (mistakes, errors, unexpected editing results) and measured their frequency. To count editing operations, inspired by the lexer phase in compilers, our measurement unit was a program token, such as a variable name, parenthesis, keyword (e.g., `void`, `for`, `if`), or semicolon. For the advanced-editing tasks, we were more interested in the strategies used. We found those by first identifying atomic editing activities and then observing their combined usage.

For all research questions, we triangulated data from the screen recordings, questionnaire, and debriefing interviews. We analyzed the questionnaire using violin plots (Likert questions) and by inspecting the open-text responses. The debriefing interviews were transcribed and inspected.

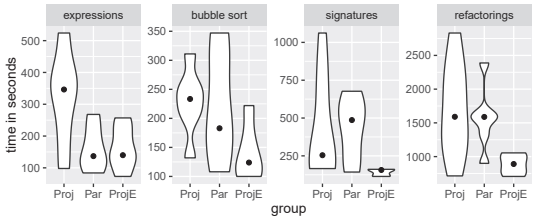


Figure 2: Task completion times in seconds (violin plots)

4. EXPERIMENT RESULTS

In total, we recorded more than 28 hours of screen recordings; collected questionnaire responses from all 19 participants; and 15 participants agreed to participate in the debriefing interviews, which amounts to 51 minutes of audio recording.

We now report our analysis results. In each subsection, we first present results for all groups together and then get back to the respective research question. Because the second phase was only a quasi experiment, we separate the discussion of the quantitative editing efficiency for RQ1 and RQ2.

4.1 Editing Efficiency

Fig. 2 shows the distributions, and Table 2 (left half) shows the average completion times for all tasks. By observing the differences between the groups, we obtained a first impression on which of our hypotheses might hold. To evaluate whether the differences are real or just occurred by chance, we conducted an ANOVA or, if its preconditions were violated, a non-parametric Kruskal-Wallis test for each task. If there was a *statistically significant* difference, we made post-hoc comparisons with adjusted p values (according to the false-discovery rate [11], to avoid alpha-level inflation due to multiple testing) to determine between which groups the *significant* difference exists. This generally leads to lower p values to reject a null hypothesis, depending on the number of rejected null hypotheses. Table 2 (right half) shows the details about the tests, together with the effect sizes (Cliff's delta [16], 95 % confidence interval).

For Task 1, the ANOVA shows a *significant* difference between the groups, but it vanishes in the post-hoc comparisons. For Task 2, there is no *significant* difference. Thus, we reject all three hypotheses for these basic-editing tasks. Yet, the effect sizes show differences, indicating that experience might even help with basic editing, which demands a valuable, future investigation, since these differences are not statistically significant. Specifically, the effect sizes again emphasize the particular difficulty when editing expressions.

Table 2: Completion times, **significance tests, effect sizes**

task	Proj[1]	Par[1]	ProjE[1]	significance tests & post-hoc comparison
1	322	162	162	ANOVA: F=4.15, df=2/13, p=.040 Adjusted p value: .017 Proj vs. Par: t=2.34, df=7.56, p=.049 Proj vs. ProjE: t=2.34, df=7.56, p=.049 ProjE vs. Par: t=0.02, df=7.99, p=.987 Cliff's delta: Proj vs. Par: 0.67; Proj vs. ProjE: 0.67; ProjE vs. Par: 0.04
2	228	224	142	ANOVA: F=2.35, df=2/13, p=.134 — Cliff's delta: Proj vs. Par: 0.13; Proj vs. ProjE: 0.73; ProjE vs. Par: -0.6
3	438	430	142	Kruskal-Wallis: χ^2=8.78, df=2, p=.012 Adjusted p value: .033 Proj vs. Par: U=23, p=.950 Proj vs. ProjE: U=40, p=.002 ProjE vs. Par: U=3, p=.030 Cliff's delta: Proj vs. Par: -0.04; Proj vs. ProjE: 1; ProjE vs. Par: -0.8
4	1700	1567	887	Kruskal-Wallis: χ^2=7.21, df=2, p=.027 Adjusted p value: .033 Proj vs. Par: U=24, p=1.000 Proj vs. ProjE: U=36, p=.019 ProjE vs. Par: U=2, p=.017 Cliff's delta: Proj vs. Par: 0; Proj vs. ProjE: 0.8; ProjE vs. Par: -0.87

[1] average time in seconds

For the other two tasks, there is a *significant* difference. The post-hoc comparison shows that, for both, the difference exists between ProjE and Proj, as well as ProjE and Par, indicating that the experts are always faster than both student groups. Thus, we accept hypotheses H2 and H3, which is supported by all effect sizes.

Let us now get back to our research questions that involve the quantitative analysis of the editing efficiency.

RQ1. In the first phase, for all tasks, we observe no *significant* difference in the completion times between projectional and parser-based editing when comparing Proj and Par:

> Editing efficiency was quickly achievable with a projectional editor. A short 45-minute training sufficed to achieve an efficiency comparable to parser-based editors.
>
> RQ1 (controlled experiment)

RQ2. Comparing results from both phases shows a *significant* difference in completion times between the basic (Tasks 1 and 2) and the advanced (Tasks 3 and 4) editing tasks. While there is no *significant* difference for basic editing, which strongly indicates that all groups had similar editing performances, ProjE outperformed Proj and even Par in advanced editing (see a discussion of the validity of this comparison in Sec. 6). Thus, experience appeared to have a substantial effect on how the advanced-editing tasks were solved.

The average projectional-editing experience of around one year that ProjE had allowed them to solve Task 3 three times (3.08) and Task 4 two times (1.92) faster than Proj. ProjE outperformed Par three times (3.03) in Task 3 and almost two times (1.77) in Task 4. This observation and our qualitative results (presented shortly in Sec. 4.2–4.3) suggest:

> In basic editing, more experience with projectional editing did not lead to *significantly* better results. In advanced editing, more experience with projectional editing led to *significantly* better results compared to projectional-editing beginners and even participants using the parser-based editor.
>
> RQ2 (quasi experiment)

4.2 Basic Editing

To address RQ3 for the first two tasks, we conducted a fine-grained analysis of editing operations and errors.

4.2.1 Editing Operations

Definitions. We considered the following four operations: *Insertion* is an attempt to add a program token, regardless of success or failure. It can lead to several tokens being added (e.g., a closing parenthesis for an opening one), which is just counted as one *Insertion*. *Deletion* is its inverse—the removal of a token. It may also trigger the removal of several tokens, which is also just counted once. *Selection* refers to highlighting one or more tokens for follow-up operations (e.g., *Deletion*). Several methods exist, such as double clicks or keyboard shortcuts, which we did not distinguish. Incremental selection (e.g., along the tree) was counted as one selection. Furthermore, if the user aborted a selection and tried it again, the retry was counted separately. *Cloning* refers to the duplication of one or more program tokens, achieved using *CopyPaste*, *CutPaste* or MPS' *Duplicate* operation.

Use of Operations. Fig. 3a and 3b show the operation use (mean per participant) in Task 1 and 2. Fig. 4 shows the respective perceptions from the post-experiment questionnaire.

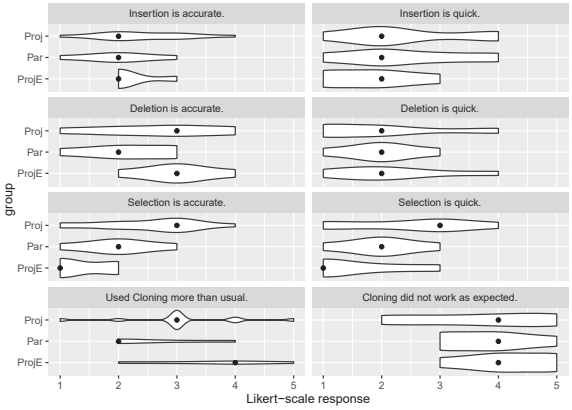
Insertion is dominant. Its ratio to the other operations

Figure 3: Average use of basic-editing operations per participant. Y-axis in log(y+1) scale.

is similar among the groups. All participants considered *Insertion* as accurate and quick: The distribution (Fig. 4) shows almost no difference between Proj and Par; ProjE was only slightly more positive. To perform *Insertions*, many participants (six in the questionnaire, three in interviews) stated that using code completion is easier than writing code character by character. The perceived need for more extensive use of code completion could lead to additional attention investment [12] of beginners—however, we notice that it did not negatively impact efficiency. In fact, writing code character by character without code completion may lead to invalid code (when tokens are not bound as typed).

We observe that *Deletion* was a bit more problematic. While the first task required more *Deletions* than the second one, ProjE was more efficient and required slightly fewer *Deletions*, given the experts' proficiency. *Deletion* was perceived as less accurate (Fig. 4) by both projectional-editor groups (Proj, ProjE), while all agreed that *Deletion* is quick. Even our beginners Proj figured out that exploiting the AST structure can be used for quick *Deletions* of whole sub-trees, such as an `if` statement together with its body: *"I used the possibility to clear the code just by going to the if statement."* The accuracy issue originated from two problems. First, the cursor placement was sometimes unpredictable: *"Cursor placement was highly influential on what got deleted, which was not always what the user expects."* Second, participants had problems with *Over-Deletions* (discussed in Sec. 4.2.2).

The use of *Selection* is a differentiator between Proj, who primarily expressed a neutral opinion about accuracy and quickness of *Selections* (confirmed by interviews), and ProjE, who were very positive and strongly agreed to both aspects. ProjE used *Selection* more often, at least in Task 1, which required changing existing code. Par generally found *Selection*



1: strongly agree, 2: agree, 3: neutral, 4: disagree, 5: strongly disagree

Figure 4: Opinions about basic editing

being accurate and quick.

Four projectional beginners (Proj) specifically expressed problems with selecting visually adjacent nodes that are not direct neighbors (only siblings) in the AST, which is not possible: *"Got some unexpected selections (they were right, just it selected not what I wanted), like in case 1+2+3, putting [the] cursor at 2, the 2+3 got selected, while wanted to select 1+2, which was not possible due to AST tree structure."*

Finally, *Cloning* was very rarely used in the first two tasks. In general, most participants found that it worked as expected. We discuss *Cloning* further in Sec. 4.3.2.

RQ3. In summary, the use of basic-editing operations did not differ much between the groups:

> Writing code was not negatively impacted by projectional editing, but relied on increased use of code completion. Selecting code required more experience and attention.
>
> RQ3 (controlled and quasi experiment)

4.2.2 Editing Errors

Definitions. From the screen recordings, we identified the following errors made repeatedly. *Invalid Insertion* is an insertion flagged as invalid (marked red) by the editor. Reasons include, for instance, the lack of an alias corresponding to the input at the current location, or an incorrect instantiation of a node. Here is an example, where a participant tried to add an invalid closing parenthesis (MPS automatically inserted the closing parenthesis after typing the opening one; adding another one between the existing parentheses is invalid):

```
void bubbleSort()) { ... }
```

We only count *Invalid Insertions* when the participant proceeded with another edit and left the previously inserted code invalid. Transitively invalid (not *yet* valid) code was not counted. For instance, entering `int` is initially not valid in mbeddr; if a user disambiguates to `int8`, it is a valid insertion (continuing with a variable name would be invalid). *Invalid Trailing Insertion* is a failed insertion that directly follows an *Invalid Insertion*. It cannot exist without an *Invalid Insertion* before (due to the absence of parsing, every single node has to be inserted step-by-step). Here is an example, where the root cause is the *Invalid Insertion* shown above:

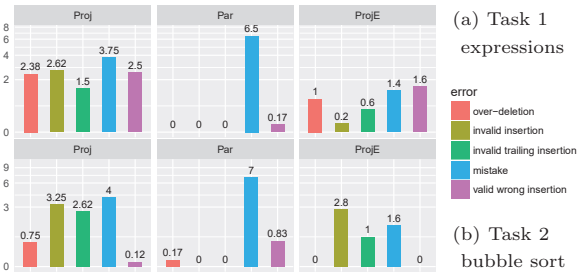
```
void bubbleSort())int8[] list, int8 n) { ... }
```

We distinguished these cases from *Invalid Insertions* to obtain more insight into edits that could potentially be correct, but were invalid, because an earlier token was not bound correctly. *Valid Wrong Insertion* is a syntactically valid insertion that is semantically wrong: an insertion was performed successfully, but the result is not what the participant expected. We count such cases irrespective of their recognition by the participant. The next code snippet gives an example, where the participant tried to negate the expression `p && q`, but entering an exclamation mark yields an incorrect result. Each line shows her steps towards the incorrect result (last line):

```
boolean deMorgansLaw = !p && q == p || q;
boolean deMorgansLaw = (!p && q == p || q;
boolean deMorgansLaw = (!p && q) == p || q;
```

Mistake is an erroneous attempt (typo) to enter one or several characters. Finally, *Over-Deletion* is a deletion that removes too many tokens. Whenever a participant deleted code and immediately reentered it, we count this as an *Over-Deletion*. We made a subjective assessment in unclear cases.

Occurrence of Errors. Fig. 5a and 5b show the occurrence of errors (mean per participant) in Task 1 and 2. Fig. 6 shows the results from the post-experiment questionnaire.



(a) Task 1 expressions

(b) Task 2 bubble sort

Figure 5: Average occurrence of errors per participant. Y-axis in log(y+1) scale.

Both Proj and ProjE made all of the identified errors—yet, the latter at a much smaller rate. The *Invalid Insertions* mainly occurred among Proj, rarely among ProjE, and never among Par. That is, in case the latter made some potentially invalid insertions—which can only be typos—they fixed them immediately. The higher ratio for Proj can then only be explained by missing awareness of the fact that nodes need to be bound immediately. We observe the same ratios for *Invalid Trailing Insertions*, which shows that developers lose time by unbound nodes. *Valid Wrong Insertions* were a significant problem for the projectional-editor groups in the expression-editing task (much less in the bubble-sort task, which required less intricate edits). Par had almost no such problems. This again confirms that editing of complex constructs (expressions) is more difficult than creating program statements (Task 2). *Mistakes* were done by all groups, but significantly more by Par. We do not observe a difference between the two tasks. This indicates that most of the typing mistakes are actually prevented by projectional editing. Finally, *Over-Deletions* appeared frequently for the projectional-editor groups. In the first task, which required many *Deletions*, they were hard to avoid and occurred for every subtask. The questionnaire (Fig. 6) and the debriefing interviews confirm *Over-Deletions*. The underlying problem was again the cursor-positioning issue. For instance, participants mentioned that a *Deletion* sometimes deleted the entire expression, or that tabbing through a parameter list, trying to delete parameters, often deleted the wrong nodes: *"Sometimes if you delete something in a parameter list, the cursor jumps to an unexpected position and you delete other stuff. Then you go back with Ctrl-Z."*

RQ3. Frequency and kinds of errors varied significantly among the groups. The differences show a tendency that Proj and ProjE performed more incorrect editings not arising from mistakes (e.g., typos), but from a wrong understanding of the technicalities of projectional editing. The higher ratios of *Valid Wrong Insertions*, *Invalid Insertions*, and *Invalid*

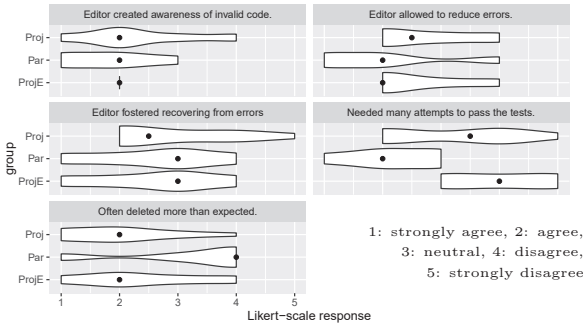


Figure 6: Opinions about errors

Trailing Insertions indicate a discrepancy between the expected editing effect and the actual result, which is especially tricky for editing expressions. In other words, participants often performed edits they assumed to be correct, but resulted in structurally incorrect code. This suggests that:

> Projectional editing showed a tradeoff between fewer *Mistakes* (e.g., typos) and an increased complexity of performing editing operations. Yet, experience reduced such problems, leading to fewer errors altogether.
>
> RQ3 (controlled and quasi experiment)

4.3 Advanced Editing

For Tasks 3 and 4, we analyzed the strategies how participants solved more advanced code-editing tasks.

4.3.1 AST Conformance

In this task, participants had to create method signatures, which also required creating the respective parameter types. This lack of predeclared `struct`s led some participants in all groups trying to find the `struct`s in the IDE by using `Ctrl-Space` and looking around in other files; eventually, all created the `struct`s. Some parser-based participants did not know how to create a valid `struct` type; some used Google to find out (see our discussion of construct validity in Sec. 6). In both projectional-editor groups, the editor-enforced syntax for `struct`s guided the users to the correct notation for `struct` definitions.

All students started by entering the method signatures (as opposed to first creating the structs). For Par, this worked well: They typed the characters and then got type-system errors that reported the missing types. While some users still struggled, all except one (he used strings for all types) eventually created the `struct`s. Proj struggled with the enforced AST conformance to various degrees. Since it is impossible to refer to `struct`s that have not yet been declared, the function signatures could not be entered before adding the `struct`s. This was perceived as problematic, but all participants found this out eventually. Yet, one struggled particularly hard. He tried to enter the signature nonetheless, typing `Book searchBook( string title )`—all of this unbound and therefore invalid. ProjE did not try to enter the signatures before the types. All of them either created the `struct`s upfront or as needed; no one tried to enter signatures while the structs were missing.

AST conformance was often mentioned in the post-experiment questionnaire and the debriefing interviews. We observe a sense of "restriction" among the participants. Program constructs need to be entered correctly the first time; a trial-and-error strategy was more challenging. Some typical programming activities felt more rigid to Proj when compared to their experience with parser-based editors. The importance of code completion was also mentioned often. While participants were positive about its effectiveness, some had ambivalent opinions and felt restricted by it. For instance, some felt they had to always use code completion to avoid invalid input: *"I do not want to use control-space all the time. I want to write my own variable and do not want to need to auto complete all the time."* In fact, while code completion tends to be used more than in text editors, many things can "just be typed" as in a text editor; yet, exceptions (e.g., when using ambiguous keywords belonging to separate languages) led to this perception.

Table 3: Usage rates of modification operations

modification operation	Proj[1]	Par[1]	ProjE[1]
Move	35 %	6 %	60 %
CopyPaste	25 %	78 %	20 %
CutPaste	75 %	56 %	53 %
RefactoringAction	30 %	11 %	53 %
ManualInsertion	45 %	56 %	33 %

[1] percentage of users in a group using the operation

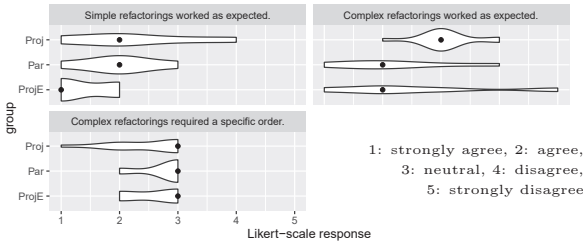


Figure 7: Opinions about refactoring operations

4.3.2 Code Modification and Refactoring

To understand code modification and refactoring strategies, we identified operations and analyzed their use in Task 4.

Definitions. The following operations were used. *Move* is the succeeded or failed transfer of one or more selected tokens to a different location using dedicated commands (e.g., Ctrl-Shift-Up/Down). *CopyPaste* and *CutPaste* are the common cloning activities. Some participants refactored by rewriting code from scratch using *ManualInsertion*. *RefactoringAction* refers to an IDE-provided refactoring invoked via the context menu or using a keyboard shortcut.

Use of Operations. For each group and each of the three subtasks, we count how many participants have used a specific operation. Table 3 shows these usage rates relative to the group size. We now discuss them per operation.

Move was used most often by ProjE followed by Proj. In Par we observe only one participant using *Move* in one subtask. For *CopyPaste* we observe the opposite: almost all of the members of Par used it, followed by Proj and ProjE. *RefactoringActions* were used mostly by ProjE followed by Proj. Only very few Par participants invoked Eclipse CDT's refactoring actions, which is again compensated by an increased use of *CopyPaste*. *ManualInsertion* is used by the majority in Par, followed by Proj, while only a third of ProjE used *ManualInsertion* to refactor code. The latter could effectively use *Move* most of the time.

Refactoring Strategies and Experiences. Based on the screen recordings, we identified the following strategies. Fig. 7 shows the respective opinions from our questionnaire.

Group Proj started with *CutPaste* when possible. If unsuccessful, they quickly switched to *ManualInsertion*. This strategy was very successful and helped them to complete all subtasks. Group Par used *CopyPaste* more often (at every possible occasion), usually followed by *ManualInsertion*, realizing a clone&own strategy. Group ProjE used *Move* whenever possible. Although a few started with *CutPaste*, they usually later switched to a *Move*-based strategy. All groups agreed that refactorings worked as expected. Yet, especially Proj felt that complex modifications had to be applied in a certain order, which was difficult to determine.

Recall the increased use of *Move* by the projectional-editor groups, for which we see two main reasons. First, experienced users are more aware of the underlying AST structure and

move code fragments along the tree to the desired position. Code can only be moved to valid locations, skipping invalid ones. This makes it a strong feature, since *Insertions* can be error-prone. Second, when using *Move* for nodes, references from other nodes do not break. With *CutPaste*, references break temporarily (and can be rebound by pressing F5). Beginners may not realize these benefits of *Move* and use *CopyPaste* and *CutPaste* instead. This is consistent with comments made in the survey and the debriefing interviews, where *CopyPaste* and *CutPaste* were perceived negatively.

Participants also commented on node references. Having to rebind such references after manual changes (e.g., *CopyPaste*) was surprising to Proj, with only some solving this issue themselves (using F5): *"When refactoring, I encountered variable names that were not discovered correctly until updated manually."* Others had to ask. This problem even led to wrong references: *"After moving code chunks into another function, some of the variables were linking to wrong objects [...]. They were not renamed all at the same time, [I] had to manually rename each of them."* Relying solely on built-in refactoring operations would have avoided this problem, however. While an automated update of references to other nodes (e.g., when renaming a variable, the variable usages are also renamed) is an advantage of projectional editing, our beginners Proj found this behavior counter-intuitive.

Finally, Proj also had problems selecting the right node for performing modifications. Often, only one node could be easily selected, but: *"It was cumbersome to understand which node to select for certain kinds of refactorings."*

RQ3. For the advanced-editing tasks, our analysis suggests:

> Projectional editing fostered a shift from *CopyPaste*- and *CutPaste*-based strategies towards operations that work well on the AST (e.g., *Move*, *Selection*). This required an increased understanding of the underlying AST.
>
> RQ3 (controlled and quasi experiment)

The decreased freedom in a projectional editor was both an advantage and a disadvantage. It guided developers when solving a task, as we specifically observed in Task 3. Yet, it enforced a specific order when performing the modifications in Task 4. This again indicates increased demand to training, practice, and understanding required for advanced editing.

5. LESSONS LEARNED

The results of our study suggest several implications for practitioners (language and tool developers) and researchers.

Quick Prototyping and Well-Designed Code. Depending on the development methodology, code evolves over time, with more and more structure added over several iterations. To sustain projectional editing in such a scenario, our results show that better code-modification facilities are needed. While programmers can become very efficient with a projectional editor when code is written from scratch, modifying existing code required more practice and understanding. More emphasis should be put on supporting quicker and more intuitive code-modification facilities, especially for expressions.

Improving Expression Editing. Editing expressions with their fine-grained tree structure is one of the major challenges in a projectional editor. The inability to insert and remove parentheses in arbitrary places (and then refactor the tree structure according to the precedence expressed by the parentheses) is an example. We suggest that future research should focus on making expression editing more efficient. Hybrid editors are one possible approach to address this challenge: they rely on on-demand linearization of (expression) tree structures. Complex-structured nodes inside expressions (e.g., tables) retain their structure and editor. After editing the linear structure, it is reparsed to reassemble the tree. The MPS team has experimented with this approach, but it is not yet clear what the trade-offs are regarding language composability, notational freedom, and editing efficiency.

References. References are based on pointers to the target node's ID. Despite some advantages (e.g., robust refactorings) of this approach, we observe problems with the tradeoff that the reference target has to exist at the time the reference is created. A more robust and intuitive handling of references is desirable. While some problems can be solved by language developers (e.g., quick fixes to create reference targets), there should be a better way to support references by the IDE itself. Recall the variable-binding problem. Participants lost time by unbound variables, which was reflected in *Invalid Trailing Insertions*—many correct insertions that are wrong (unbound) due to one invalid *Insertion* before.

6. THREATS TO VALIDITY

Internal Validity. The validity of RQ2's results for hypothesis H2 is limited, since the industrial participants were not randomly split into a parser and projectional-editor group. However, our experience shows that they are equally proficient with textual editing (i.e., experts in using a keyboard) as the graduate CS students. Furthermore, only basic programming skills (knowledge of program structure) were required, not creating algorithms or reasoning about logical laws. This mitigated any potential differences in programming experience among the participants. Students are also known to perform like industry participants in similar conditions [23, 40], which is supported by the similar editing efficiencies of Proj and Par in basic editing.

To assure the quality of our tasks and materials, we relied on the experience of our second author (who led mbeddr's development) and two think-aloud pilots [47] done by two other authors: writing a *Hello-World* program (from mbeddr's user guide [49]) and creating a calculator language (an MPS tutorial [7]). The first pilot showed that new users can cope with MPS, but providing a startup package is crucial. The second pilot helped us create this package and the tasks. With these improvements, we also pilot-tested the experiment with volunteer students. To avoid distraction by IDE specifics, a 45-minute training session ensured a basic understanding of the IDE, the startup package, navigating menus/files, and running each task's test case. Furthermore, the students had used Eclipse during their studies, and Eclipse CDT is similar (including menu locations of refactorings).

External Validity. The experiment tasks might not completely reflect practitioners' everyday code-editing tasks. We carefully designed them based on the survey, covering a wide range of common code-editing activities, including writing and modifying code at different granularity levels. Still, the overall editing efficiency depends on the relative distribution of the evaluated activities, so transferring our results to tasks with different structures needs to be done with care.

A limitation is that our experiment is based on one projectional editor only. Yet, MPS is a substantial case, as it is currently the most mature tool with the largest user base.

Finally, measuring the degree of understanding or the

mental overhead (attention investment [12]) was beyond the scope of our study, but would be valuable future work.

Construct Validity. For RQ1, we measured efficiency using completion time, which is a reliable, low-level metric. It is also a valid metric, since we subtracted any distractions of the participants (e.g., when using the browser or Skype), which happened rarely (1 to 2 participants per group), and we made sure that the tasks were finished and either had a completely correct solution or one that only slightly deviated.

For RQ3 we triangulated various measurements. We watched screen recordings to identify operations and their frequency, cross-checking these results on random samples. We also inspected the recordings for strategies and patterns. To increase reliability of this analysis, we created short write-ups of the strategy applied per participant and task. Finally, we analyzed the questionnaire's Likert questions using violin plots—interpreting the Likert scale as a continuous scale, which is justified given equal distances between the values.

Statistical Conclusions Validity. We mitigated different group sizes using the robust tests ANOVA and Kruskal-Willis.

7. RELATED WORK

We discuss the editing efficiency of projectional editors, followed by related general studies of programming efficiency.

Projectional Editors from the 1980s. GANDALF [35] and the Incremental Programming Environment (IPE) [32] do not attempt to make editing textual notations efficient; for example, they lack support for linear editing of tree-structured expressions. The Synthesizer Generator [39] avoids the use of projectional editing at the fine-grained expression level, where textual input and parsing is used. While this may improve editing efficiency, it risks the advantages of projectional editing, because language composition at the expression level is limited. Another work that implements and uses a DSL within the Synthesizer Generator [37] concludes: "Program editing will be considerably slower than normal keyboard entry, although actual time spent programming non-trivial programs should be reduced due to reduced error rates." Our study confirms that certain kinds of errors can be reduced, but finds that efficient editing is possible.

Contemporary Projectional Editors. For all of the following projectional editors, our results can be used to improve their efficiency. The Intentional Domain Workbench (IDW) is the most recent implementation of the Intentional Programming paradigm [44, 18], supporting diverse notations [45, 14]. Since it is a commercial, closed-source project without widespread adoption yet, we cannot easily study it or survey its users. Clark describes a projectional editor [15] that uses term rewriting to create the concrete from the abstract syntax. It supports graphical and textual notations, but is in an early stage and lacks support for efficient editing of text.

The language-workbench comparison by Erdweg et al. [20] does not use editing efficiency as a comparison criterion. We address this gap, so we can relate our results and MPS' capabilities to the three analyzed projectional editors: Más, Onion, and Whole Platform. Más, using the Concrete editor [3], supports a range of notations relying on HTML/CSS. Yet, for editing textual notations efficiently, only side transformations [55] are available. Onion does not use projection for textual notations, so our results are not relevant. The Whole Platform [9] emphasizes structured notations, but has no further support for efficiently editing structured text.

Editing and Programming Efficiency. Ko et al. [26] study how programmers edit Java code in an experiment similar to ours, where programmers performed editing tasks. They identify *"a set of patterns that suggest that the full flexibility of unstructured text is not required for most of the modifications that programmers make to code."* They propose these patterns as editing primitives in editors. Although their list of patterns is richer, some are overlapping with what we found. For instance, creating a method call by code completion was used very frequently. Their findings might explain the efficiency of projectional editing we observed. Studying editing patterns at an even finer granularity, aligning our results with theirs, would be valuable future work.

Scratch is a visual environment for learning programming. It uses a projectional editor, but does not focus on textual editing. Still, Lewis' experiment [28] comparing attitudinal and learning outcomes of students programming in Scratch compared to students using the text-based Logo shows that enriching textual programming with visual information is beneficial. This in fact also motivates projectional editing.

Miller et al. [33] propose, implement, and evaluate multiple selections in a text editor. While the results on multiple selection are not related to our work, their evaluation of the efficiency of using multiple selections also relies on two groups of students who performed editing tasks under supervision.

Poller et al. [36] compare a moded (vi) and modeless (emacs) editor in an experiment designed similarly to ours. The participants performed text-editing tasks, analyzed according to task completion time and errors, complemented by a questionnaire. As a result, the modeless editor allowed editing more freely, but led to more errors and slightly reduced editing efficiency. The moded editor was better for relatively fixed tasks. Compared to our experiment, it is striking that less freedom, although of very different nature, also led to higher efficiency. Yet, while the authors propose combining moded and modeless editors, our results do not suggest combining projectional and parser-based editors.

8. CONCLUSION

We presented an experiment that investigated the effects of projectional editing on editing efficiency, editing strategies, and types and frequencies of errors. The experiment was conducted with 19 participants: graduate CS students and industrial developers performing common code-editing tasks.

Our results show that efficiency with projectional editing can be quickly achieved for basic code-editing activities. More experience does not lead to better results. In contrast, advanced editing (e.g., larger code modifications or refactorings) requires significantly more experience and understanding of the underlying concepts (in particular, the AST structure). The projectional editor fosters fewer errors (mistakes) and different editing strategies (e.g., increased use of operations that work well on ASTs). Optimizing towards these strategies could further increase the efficiency.

We plan to work with the tool developers (in particular, the MPS team) to categorize and address the hotspots discovered in our study (including expression editing, reference binding, and AST-based selections), find generalized solutions, and develop an improved editing framework for projectional editors. Finally, our study is limited to editing efficiency. Systematically measuring the impact of arbitrary language composition (i.e., language embedding) and flexible notations (graphical and textual) is valuable future work.

9. REFERENCES

[1] CamStudio Desktop Screen Recorder. http://sourceforge.net/projects/camstudio.

[2] Code Orchestra IDE. http://codeorchestra.com/ide.

[3] Concrete. http://concrete-editor.org.

[4] Jetbrains MPS. http://www.jetbrains.com/mps.

[5] mbeddr. http://mbeddr.com.

[6] mbeddr Graphical State Machines. http://mbeddr.com/2015/03/05/graphicalSM.html.

[7] MPS Tutorial. http://www.jetbrains.com/mps/docs/tutorial.html.

[8] Online Appendix. http://gsd.uwaterloo.ca/projectional-workbenches.

[9] Whole Platform. http://whole.sourceforge.net.

[10] T. W. Anderson and J. D. Finn. *The New Statistical Analysis of Data*. Springer, 1996.

[11] Y. Benjamini and Y. Hochberg. Controlling the false discovery rate: A practical and powerful approach to multiple testing. *Journal of the Royal Statistical Society Series B (Methodological)*, 57(1):289–300, 1995.

[12] A. F. Blackwell. First steps in programming: A rationale for attention investment models. In *Proc. HCC*, 2002.

[13] M. Broy, S. Kirstan, H. Krcmar, and B. Schätz. What is the Benefit of a Model-Based Design of Embedded Software Systems in the Car Industry? In J. Rech and C. Bunse, editors, *Emerging Technologies for the Evolution and Maintenance of Software Models*. IGI Global, 2011.

[14] M. Christerson and H. Kolk. Domain expert DSLs, 2009. talk at QCon London 2009, available at http://www.infoq.com/presentations/DSL-Magnus-Christerson-Henk-Kolk.

[15] T. Clark. A Declarative Approach to Heterogeneous Multi-Mode Modelling Languages. In *Proc. GEMOC*, 2014.

[16] N. Cliff. Dominance statistics: Ordinal analyses to answer ordinal questions. *Psychological Bulletin*, 114(3):494, 1993.

[17] J. O. Coplien. *Multi-paradigm Design for C++*. Addison-Wesley, 1999.

[18] K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[19] S. Erdweg, P. G. Giarrusso, and T. Rendel. Language Composition Untangled. In *Proc. LDTA*, 2012.

[20] S. Erdweg, T. van der Storm, M. Völter, M. Boersma, R. Bosman, W. R. Cook, A. Gerritsen, A. Hulshout, S. Kelly, A. Loh, et al. The State of the Art in Language Workbenches. In *Proc. SLE*, 2013.

[21] M. Fowler. Language Workbenches: The Killer-App for Domain Specific Languages? http://www.martinfowler.com/articles/languageWorkbench.html, 2005.

[22] J. Gray and G. Karsai. An Examination of DSLs for Concisely Representing Model Traversals and Transformations. In *Proc. HICSS*, 2003.

[23] M. Höst, B. Regnell, and C. Wohlin. Using Students As Subjects—A Comparative Study of Students and Professionals in Lead-Time Impact Assessment. *Empirical Softw. Engg.*, 5(3):201–214, Nov. 2000.

[24] J. Hutchinson, J. Whittle, M. Rouncefield, and S. Kristoffersen. Empirical Assessment of MDE in Industry. In *Proc. ICSE*, 2011.

[25] R. B. Kieburtz, L. McKinney, J. M. Bell, J. Hook, A. Kotov, J. Lewis, D. P. Oliva, T. Sheard, I. Smith, and L. Walton. A Software Engineering Experiment in Software Component Generation. In *Proc. ICSE*, 1996.

[26] A. J. Ko, H. H. Aung, and B. A. Myers. Design Requirements for More Flexible Structured Editors from a Study of Programmers' Text Editing. In *Proc. CHI EA*, 2005.

[27] A. Kuhn, G. C. Murphy, and C. A. Thompson. An Exploratory Study of Forces and Frictions Affecting Large-Scale Model-Driven Development. In *Proc. MODELS*. 2012.

[28] C. M. Lewis. How programming environment shapes perception, learning and goals: Logo vs. scratch. In *Proc. SIGCSE*, 2010.

[29] G. Liebel, N. Marko, M. Tichy, A. Leitner, and J. Hansson. Assessing the state-of-practice of model-based engineering in the embedded systems domain. In *Proc. MODELS*, 2014.

[30] P. Liggesmeyer and M. Trapp. Trends in Embedded Software Engineering. *IEEE Softw.*, 26(3):19–25, May 2009.

[31] M. Lillack, T. Berger, and R. Hebig. Experiences from reengineering and modularizing a legacy software generator with a projectional language workbench. In *Proc. SPLC*. 2016.

[32] R. Medina-Mora and P. H. Feiler. An Incremental Programming Environment. *IEEE Trans. Softw. Eng.*, 7(5):472–482, Sept. 1981.

[33] R. C. Miller and B. A. Myers. Multiple Selections in Smart Text Editing. In *Proc. IUI*, 2002.

[34] Z. Molotnikov, M. Völter, and D. Ratiu. Automated domain-specific C verification with mbeddr. In *Proc. ASE*, 2014.

[35] D. Notkin. The GANDALF Project. *J. Syst. Softw.*, 5(2):91–105, May 1985.

[36] M. F. Poller and S. K. Garter. A Comparative Study of Moded and Modeless Text Editing by Experienced Editor Users. In *Proc. CHI*, 1983.

[37] S. W. Porter. Design of a Syntax Directed Editor for PSDL (Prototype Systems Design Language). Master's thesis, Naval Postgraduate School, Monterey, CA, USA, 1988.

[38] D. Ratiu, B. Schaetz, M. Voelter, and B. Kolb. Language engineering as an enabler for incrementally defined formal analyses. In *Proc. FormSERA*, 2012.

[39] T. W. Reps and T. Teitelbaum. The Synthesizer Generator. In *Proc. SDE*, 1984.

[40] P. Runeson. Using Students as Experiment Subjects—An Analysis on Graduate and Freshmen Student Data. In *Proc. EASE*, 2003.

[41] B. Selic. The Pragmatics of Model-Driven Development. *IEEE Softw.*, 20(5):19–25, Sept. 2003.

[42] W. Shadish, T. Cook, and D. Campbell. *Experimental and Quasi-Experimental Designs for Generalized Causal Inference*. Houghton Mifflin Company, 2002.

[43] M. Simi and F. Campagne. Composable Languages for Bioinformatics: The NYoSh Experiment. *PeerJ*, 2:e241, 2014.

[44] C. Simonyi. The death of computer languages, the birth of intentional programming. In *Proc. NATO Science Committee Conference*, 1995.

[45] C. Simonyi, M. Christerson, and S. Clifford. Intentional Software. In *Proc. OOPSLA*, 2006.

[46] A. Strauss and J. Corbin. Open Coding. *Basics of Qualitative Research: Grounded Theory Procedures and Techniques*, 2:101–121, 1990.

[47] M. W. Van Someren, Y. F. Barnard, J. A. Sandberg, et al. *The Think Aloud Method: A Practical Guide to Modelling Cognitive Processes*. Academic Press London, 1994.

[48] E. Visser. WebDSL: A Case Study in Domain-Specific Language Engineering. In R. Lämmel, J. Visser, and J. a. Saraiva, editors, *Generative and Transformational Techniques in Software Engineering II*, pages 291–373. Springer, 2008.

[49] M. Voelter. *mbeddr C User Guide*. Itemis AG. http://mbeddr.com/userguide/UserGuideExport.html.

[50] M. Voelter. Language and IDE Modularization and Composition with MPS. In *GTTSE*, LNCS. Springer, 2011.

[51] M. Voelter and S. Lisson. Supporting Diverse Notations in MPS' Projectional Editor. 2014.

[52] M. Voelter, D. Ratiu, B. Kolb, and B. Schätz. mbeddr: Instantiating a Language Workbench in the Embedded Software Domain. *Autom. Softw. Eng.*, 20(3):339–390, 2013.

[53] M. Voelter, D. Ratiu, B. Schaetz, and B. Kolb. mbeddr: an extensible c-based programming language and ide for embedded systems. In *Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity*, pages 121–140. ACM, 2012.

[54] M. Voelter, D. Ratiu, and F. Tomassetti. Requirements as First-Class Citizens. In *Proc. ACES-MB*, 2013.

[55] M. Voelter, J. Siegmund, T. Berger, and B. Kolb. Towards User-Friendly Projectional Editors. In *Proc. SLE*, 2014.

[56] M. Voelter, A. van Deursen, B. Kolb, and S. Eberle. Using C Language Extensions for Developing Embedded Software: A Case Study. In *Proc. OOPSLA*, 2015.

[57] M. Voelter, A. van Deursen, B. Kolb, and S. Eberle. Using c language extensions for developing embedded software: A case study. In *Proc. OOPSLA*, 2015.